AFRL-RI-RS-TR-2009-154
Final Technical Report
June 2009

POLLUX: ENHANCING THE QUALITY OF SERVICE OF THE GLOBAL INFORMATION GRID (GIG)

The Vanderbilt University

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2009-154 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/
NORMAN AHMED
Work Unit Manager

/s/
JAMES W. CUSACK, Chief
Information Systems Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JUN 2009	Final	Feb 2006 – Feb 2009

4. TITLE AND SUBTITLE

POLLUX: ENHANCING THE QUALITY OF SERVICE OF THE GLOBAL INFORMATION GRID (GIG)

5a. CONTRACT NUMBER
FA8750-06-2-0054

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62702F

6. AUTHOR(S)

Douglas C. Schmidt

5d. PROJECT NUMBER
ICED

5e. TASK NUMBER
06

5f. WORK UNIT NUMBER
05

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

The Vanderbilt University
2015 Terrace Place
Nashville, TN 37203

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RISE
525 Brooks Rd.
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
N/A

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2009-154

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW-2009-2417

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The Pollux project focused on developing, analyzing, empirically evaluating, and optimizing technologies that can support the GIG's real-time QoS needs. This effort also focused on solutions that leveraged and enhanced commercial-off-the shelf (COTS) and standards-based technologies. The final report describes the results of the Pollux project during the period of April 2006 to February 2009. Pollux focused on the following primary technical focus areas during this period of performance: DDS Benchmarking Environment (DBE), DDS QoS Modeling Language (DQML), Resource Allocation and Control Engine (RACE), Model-driven engineering tools for QoS configuration, Ricochet++ adaptive middleware/transport framework, CUTS System Execution Modeling Tool.

15. SUBJECT TERMS
Publish and subscribe, Quality of Service (QoS), Information Management, distributed computing.

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Norman Ahmend
U	U	U	UU	31	19b. TELEPHONE NUMBER (*Include area code*) N/A

Table of Contents

Table of Figures

LIST OF TABLES

1.0 SUMMARY

Future DoD missions will run on *system of systems* (SoS) characterized by thousands of platforms, sensors, decision nodes, weapons, and warfighters connected through heterogeneous wireline and wireless networks to exploit information superiority and achieve strategic and tactical objectives. The networks, operating systems, middleware, and applications in SoSs offer a combinatoric number of configuration points for adjusting their resource requirements and the quality of service (QoS) they deliver. The Global Information Grid (GIG) is an emerging DoD SoS intended to organize and coordinate this large application and technology space to manage information effectively and provide DoD planners and warfighters with the right information to the right place at the right time. To successfully support enterprise and tactical information management needs, the emerging GIG SoS technologies must provide (1) universal–yet secure–access to information from a wide variety of sources running over a wide variety of hardware/software platforms and networks, (2) an orchestrated information environment that aggregates, filters, and prioritizes the delivery of this information to work effectively in the face of transient and enduring resource constraints, (3) continuous adaptation to changes in the operating environment, such as dynamic network topologies, publisher/subscriber (pub/sub) membership changes, and intermittent connectivity, and (4) tailorable, actionable information that can be distributed in a timely manner in the appropriate form and level of detail to users at all echelons.

The Pollux project was a 36 month R&D project focused on developing, analyzing, empirically evaluating, and optimizing technologies that can support the GIG's real-time QoS needs. This effort also focused on solutions that leveraged and enhanced commercial-off-the shelf (COTS) and standards-based technologies. This final report describes the results of the Pollux project during the period of April 2006 to February 2009. As described below, Pollux focused on the following primary technical focus areas during this period of performance:

1. The **DDS Benchmarking Environment (DBE)**, which enabled the precise analysis of the latency, jitter, and throughput of standard-based and/or COTS-based QoS-enabled pub/sub technologies, including DDS, JMS, Web Services, and CORBA.
2. The **DDS QoS Modeling Language (DQML)**, which enables developers of QoS-enabled pub/sub SoS to specify and enforce QoS policies that capture user intents and ensuring the preservation of information priorities and differentiated flows of information securely and predictably through the GIG
3. The **Resource Allocation and Control Engine (RACE)**, which is a middleware framework that supports predictable and scalable GIG application performance, even in the face of changing operational conditions, workloads, and resource availability.
4. **Model-driven engineering tools for QoS configuration** that developers of GIG SoSs can use to bind of application-level QoS policies onto the solution space comprising the QoS mechanisms for tuning the underlying middleware.
5. The **Ricochet++ adaptive middleware/transport framework** that integrates QoS-enabled pub/sub middleware (such as DDS) with the Ricochet transport protocol developed by Cornell as part of the Castor project.
6. Enhancements to the **CUTS System Execution Modeling Tool** that enables developers conduct "what if" experiments to discover, measure, and rectify performance problems *early* in the lifecycle (e.g., in the architecture and design phases), as opposed to the integration phase, when mistakes are much harder and more costly to fix.

The remainder of this final report summarizes our results in each of these technical focus areas.

2.0 THE DDS BENCHMARKING ENVIRONMENT (DBE)

Tactical information management systems increasingly run in net-centric environments characterized by thousands of platforms, sensors, decision nodes, and computers connected together to exchange information, support sense-making, enable collaborative decision making, and effect changes in the physical environment. For example, the Global Information Grid (GIG) is an ambitious net-centric environment being designed to ensure that different services and coalition partners, as well as individuals participating to specific missions, can collaborate effectively and deliver appropriate firepower, information, or other essential assets to warfighters in a timely, dependable, and secure manner. Achieving this vision requires the following capabilities from the distributed middleware software:

- **Shared operational picture.** A key requirement for mission-critical net-centric systems is the ability to share an operational picture with planners, warfighters, and operators in real-time.
- **Ensure the right data gets to the right place at the right time** by satisfying end-to-end quality of service (QoS) requirements, such as latency, jitter, throughput, dependability, and scalability.
- **Interoperability and portability in heterogeneous environments** since net-centric systems are faced with unprecedented challenges in terms of platform and network heterogeneity.
- **Support for dynamic coalitions**. In many net-centric tactical information management systems, dynamically formed coalition of nodes will need to share a common operational picture and exchange data seamlessly.

Prior middleware technologies, such as the Common Object Request Broker Architecture (CORBA) Event Service and Notification Service, and the Java Message Service (JMS), and various other proprietary middleware product, have historically lacked key architectural and QoS capabilities, such as dependability, survivability, scalability, determinism, security, and confidentiality, needed by net-centric systems for tactical information management. To address these limitations—and to better support tactical information management in net-centric systems like the GIG—the OMG has adopted the Data Distribution Service (DDS) specification, which is a standard for QoS-enabled data-centric publish/subscribe (pub/sub) communication aimed at net-centric tactical information management systems. DDS is used in a wide range of military and commercial systems.

We developed a DDS Benchmarking Environment (DBE) to facilitate testing of DDS products to determine their suitability for tactical information management. In this part of our Pollux project we collected results from many new benchmarks that compared the performance of DDS implementations in various configurations. We also evaluated the portability and configuration details of each DDS implementation.

The DDS Benchmarking Environment (DBE) consists of

- A directory structure to organize scripts, config files, test ids, and test results
- A hierarchy of Perl scripts to automate test setup and execution
- A tool for automated graph generation
- A shared library for gathering results and calculating statistics

The DBE testbed directory structure is organized as follows:

- Settings
  - Network: ~/DDS/settings/net/
  - QoS: ~/DDS/settings/qos/
  - Test Id: ~/DDS/settings/id.gen
- Start Script: ~/DDS/scripts/benchmark.pl
- Results:
  - Individual Tests: ~/DDS/results/#id/
  - Test List File: ~/DDS/results/tests.list

The DBE has the following three levels of execution:

- *First level*: This level provides the user interface and includes the following files:
  benchmark.pl →start_sub.pl
  →start_pub.pl
  →start_repo.pl
- *Second level*: This level manipulates the node itself and includes the following files:
  - start_sub.pl →executable (*i.e*. subscriber.exe)
  - start_pub.pl →executable (*i.e* publisher.exe)
  - start_repo.pl →executable (*i.e*. repo.exe)
- *Third level*: This level comprises the actual executables, e.g., publishers and subscribers written for NDDS from RTI (DDS1), OpenSPlice from PrismTechnologies (DDS2), and Open DDS (DDS3) from OCI.

We ran experiments using a simple byte-sequence data type of varying lengths, measuring throughput for various values of some other parameter such as number of subscribers, multicast vs unicast, or subscriber notification via listener (asynchronous) vs wait-on-condition (synchronous). Figure 1 is a typical example, where we compared average subscriber throughput (1 publisher and 12 subscribers) for a single DDS implementation using unicast vs multicast.

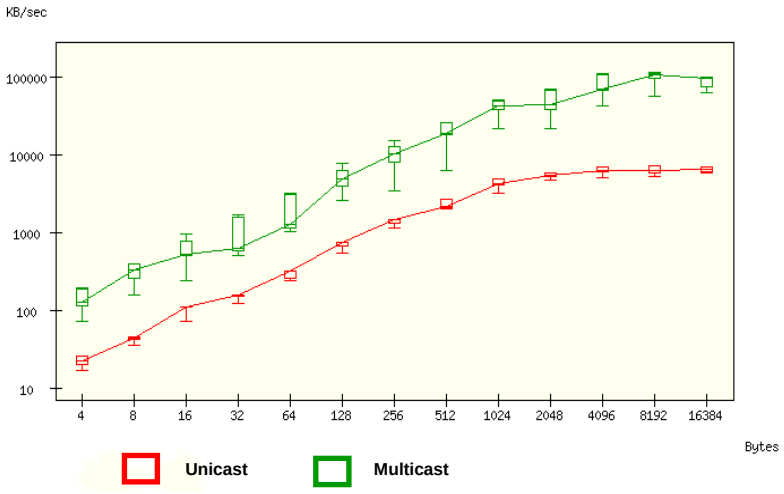


Figure 1: Unicast vs Multicast with 1 Publisher and 12 Subscribers

In Figure 2 we compare two DDS implementations, each one using multicast with 1 publisher and 12 susbscribers. The third implementation of DDS we are testing does not support multicast, so we could not get an apples-to-apples comparison with this implementation.

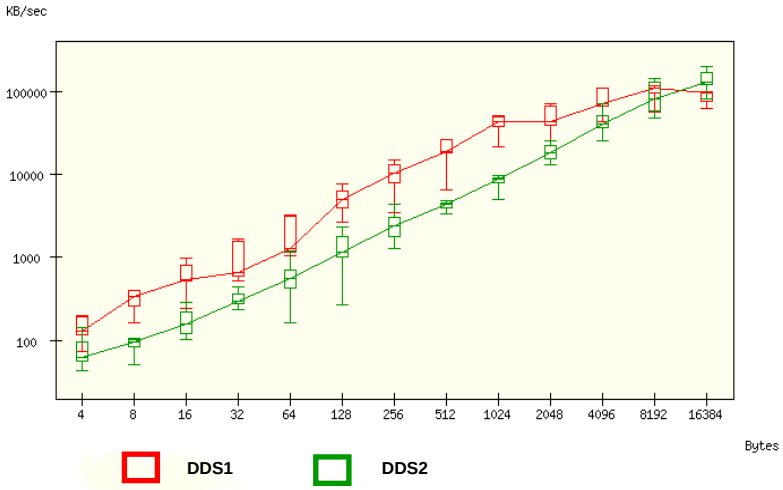
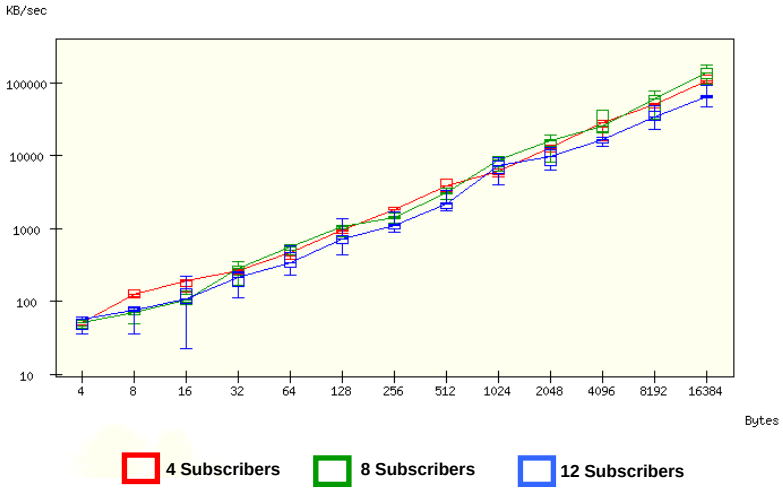


Figure 2: DDS1 vs DDS2 with 1 Publisher and 12 Subscribers (Multicast)

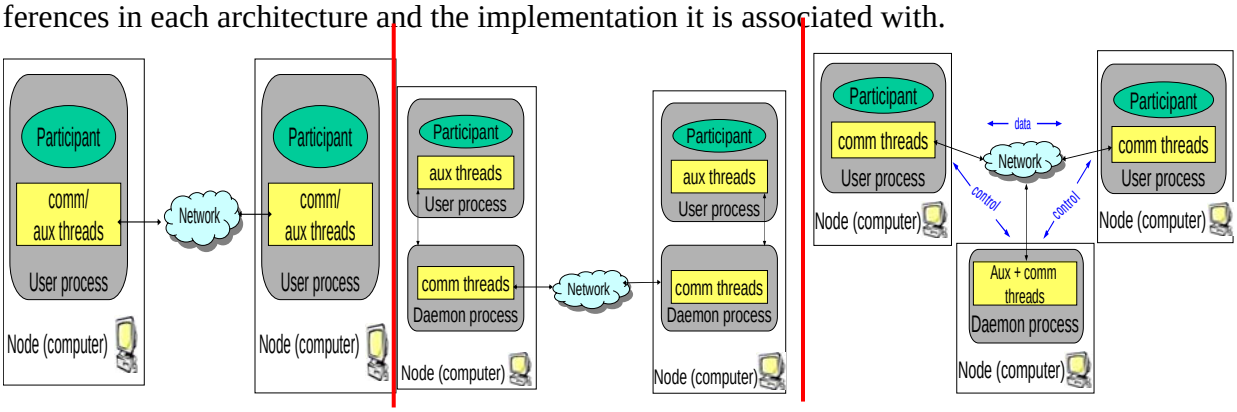
In Figure 3 we compare average subscriber throughput as we scale up the number of subscribers, on a single DDS implementation using broadcast.



Figure

Figure 3: Scaling Up Subscribers Using Broadcast

Evaluating the performance of the DDS implementations overall, we found that each has a very distinct architecture, and that the design decisions that led to these architectures have a clear effect on the conditions under which each implementation performs best. Figure 4 shows the differences in each architecture and the implementation it is associated with.



DDS1 - Decentralized Architecture **DDS2 – Federated Architecture** **DDS3 – Centralized Architecture**

Figure 4: DDS Architectures

In general, DDS1 is the best performer with smaller payload sizes, while DDS2 seems to scale better, both to larger payloads and to larger numbers of subscribers (we intend to scale our tests to both larger payloads and more subscribers per publisher to see if this trend continues). DDS3 is much newer and, while it lags behind the other two both in performance and in supported specification features, it is open-source and has a pluggable transport framework, giving it the potential for rapid improvement. However, its centralized control architecture suggests scalability problems, especially with the number of subscribers, and indeed we found this to be the case.

We also evaluated how easily a DDS application could be ported from one DDS implementation to another. Since the DDS specification is a relatively young one, we found some issues here, as expected. Figure 5 shows the most significant challenges encountered. In some cases, the feature in question is simply underspecified, and the implementation has no choice but to put a proprietary solution in place. In other cases, however, proprietary features and mechanisms have been used to facilitate performance or discovery optimization.

	DDS1	DDS2	DDS3
DomainParticipantFactory	compliant	compliant	proprietary function
Register Data Types	static method	member method	member method
Spec Operations	extra argument (newer spec)	compliant	compliant
Key Declaration	//@key	single #pragma	pair of #pragma
Required App. IDs	publisher & subscriber	none	publisher
Required App. Transport Config	code-based	none	file-based or code-based

Figure 5: DDS Portability Challenges

The remainder of this section describes the challenges we encountered conducting the experiments presented above and summarizes the lessons learned from our efforts.

2.1 Challenge 1: Synchronizing Distributed Clocks

Problem. It is hard to precisely synchronize clocks among applications running on blades distributed throughout ISISlab. Even when using the Network Time Protocol (NTP), we still experienced differences in time that led to inconsistent results and forced us to constantly repeat the synchronization routines to ensure the time on different nodes was in sync. We therefore needed to avoid relying on synchronized clocks to measure latency, jitter, and throughput.

Solution. For our latency experiments, we have the subscriber send a minimal reply to the publisher, and use on the clock on the publisher side to calculate the roundtrip time. For throughput, we use the subscriber's clock to measure the time required to receive a designated number of samples. Both methods provide us with common reference points and minimize timing errors through the usage of effective latency and throughput calculations based on a single clock.

2.2 Challenge 2: Automating Test Execution

Problem. Achieving good coverage of a test space where parameters can vary in several orthogonal dimensions leads to a combinatorial explosion of test types and configurations. Manually running tests for each configuration and each middleware implementation on each node is tedious, error-prone, and time-consuming. The task of managing and organizing test results also grows exponentially along with the number of distinct test configuration combinations.

Solution. The DBE stemmed from our efforts to manage the large number of tests and the associated volume of result data. Our efforts to streamline test creation, execution and analysis are ongoing, and include work on several fronts, including a hierarchy of scripts, several types of configuration files, and test code refactoring.

2.3 Challenge 3: Handling Packet Loss

Problem. Since our DDS implementations use the UDP transport, packets can be dropped at the publisher and/or subscriber side. We therefore needed to ensure that the subscribers get the designated number of samples despite packet loss.

Solution. One way to solve this problem is to have the publisher send the number of messages subscribers expect to receive and then to stop the timer when the publisher is done. The subscriber could then use only the number of messages that were actually received to calculate the throughput. However, this method has two drawbacks: (1) the publisher must send extra notification messages to stop the subscribers, but since subscribers may not to receive this notification message the measurement would never happen and (2) the publisher stops the timer, creating a distributed clock synchronization problem discussed in Challenge 1 that could affect the accuracy of the evaluation. To address these drawbacks we therefore adopted an alternative that ensures subscribers a deterministic number of messages by having the publishers "oversend" an appropriate amount of extra data.. With this method, we avoid extra "pingpong" communication between publishers and subscribers. More importantly, we can measure the time interval entirely at the subscriber side without relying on the publisher's clock. The downside of this method is that we had to conduct experiments to determine the appropriate amount of data to oversend.

2.4 Challenge 4: Ensuring Steady Communication State

Problem. Our benchmark applications must be in a steady state when collecting statistical data.

Solution. We send primer samples to "warm up" the applications before actually measuring the data. This warmup period allows time for possible discovery activity related to other subscribers to finish, and for any other first-time actions, on-demand actions, or lazy evaluations to be completed, so that their extra overhead does not affect the statistics calculations.

2.5 Summary of Lessons Learned

Based on our test results, experience developing the DBE, and numerous DDS experiments, we learned the following:

- **DDS Performs significantly better than other pub/sub implementations.** Our results that even the slowest DDS was about twice as fast as non-DDS pub/sub services. We also showed that DDS pub/sub middleware scales better to larger payloads compared to non-DDS pub/sub middleware. This performance margin is due in part to the fact that DDS decouples the information intent from information exchange. In particular, XML-based pub/sub mechanisms, such as SOAP, are optimized for transmitting strings, whereas the data types we used for testing were sequences. GSOAP's poor performance with large payloads is due to the fact that GSOAP (de)marshals each element of a sequence, which may be as small as a single byte, while DDS implementations send and receive these data types as blocks.
- **Individual DDS architectures and implementations are optimized for different use cases**. Our results showed that NDDS's decentralized architecture is optimized for smaller payload sizes compared to OpenSplice's federated architecture. As payload size increases, especially for the complex date types, OpenSplice catches up and surpasses NDDS in performance on the same blade. When the publisher and subscriber run on different blades, however, NDDS outperforms OpenSplice for all tested data sizes.
- **Apples-to-apples comparisons of DDS implementations are hard**. The reasons for this difficulty fall into the following categories: (1) *no common transport protocol* – the DDS implementations that we investigated share no common application protocol, *e.g.*, RTTI uses a RTPS-like protocol on top of UDP, OpenSplice will add RTPS support soon, and TAO DDS simply implements raw TCP and UDP, (2) *no universal support for unicast/broadcast/multicast* – Table 1 shows the different mechanisms supported by each DDS implementations, from which we can see DDS3 does not support any group communication transport, making it hard to maintain performance as the number of subscribers increases, (3) *DDS applications are not yet portable*, which stem partially from the fact that the specification is still evolving and vendors use proprietary techniques to fill the gaps (a portability wrapper façade would be a great help to any DDS application developer, and a huge help to our efforts in writing and running large numbers of benchmark tests), and (4) *arbitrary default settings for DDS implementations,* which includes network-specific parameters not covered by the DDS specifications that can significant impact performance.

Table 1: Supported DDS Communication Models

Impl	Unicast	Multi-cast	Broadcast
DDS1	Yes (default)	Yes	No
DDS2	No	Yes	Yes (default)
DDS3	Yes (default)	No	No

More test results, as well as more information about the tests, DBE testbed, implementation architectures, and portability issues can be found at

http://www.dre.vanderbilt.edu/DDS

We presented the results of our DBE experiments at the Object Management Group's Real-time and Embedded Systems Workshop in Ballston, VA on July 11th, 2006 and the Defense Transformation and Net-Centric Systems conference, April 9-13, 2007, Orlando, Florida.

3.0 THE DDS QOS MODELING LANGUAGE (DQML)

We developed the DDS Quality of Service (QoS) Modeling Language (DQML), which facilitates the design of QoS configurations for DDS applications and provides constraint checking to support "correct by construction" QoS configurations. DQML checks for compatibility constraint errors where data will not flow between DDS DataReaders and DataWriters due to incompatible QoS settings on the entities. It also checks for consistency constraint errors where multiple QoS settings for a particular entity will not be used because they conflict with each other.

DQML was developed using the Generic Modeling Environment (GME) tool. DQML models are also developed in GME using the DQML paradigm. GME provides a GUI for interacting with DQML so that DDS entity and QoS policy icons can be dragged and dropped onto a design space. Connections can then be made between DDS entities, as well as between DDS entities and QoS policies. Additionally, a DQML interpreter was developed to create QoS settings files for the DDS Benchmarking Environment (DBE). Once a DQML model has been developed, the modeler can then invoke the DBE interpreter to generate QoS settings files for the DataReaders and DataWriters that DBE will deploy. DBE can directly read and use these files with no manual intervention needed with respect to the QoS settings. Figure 6 illustrates the various elements and their relationships in DQML.

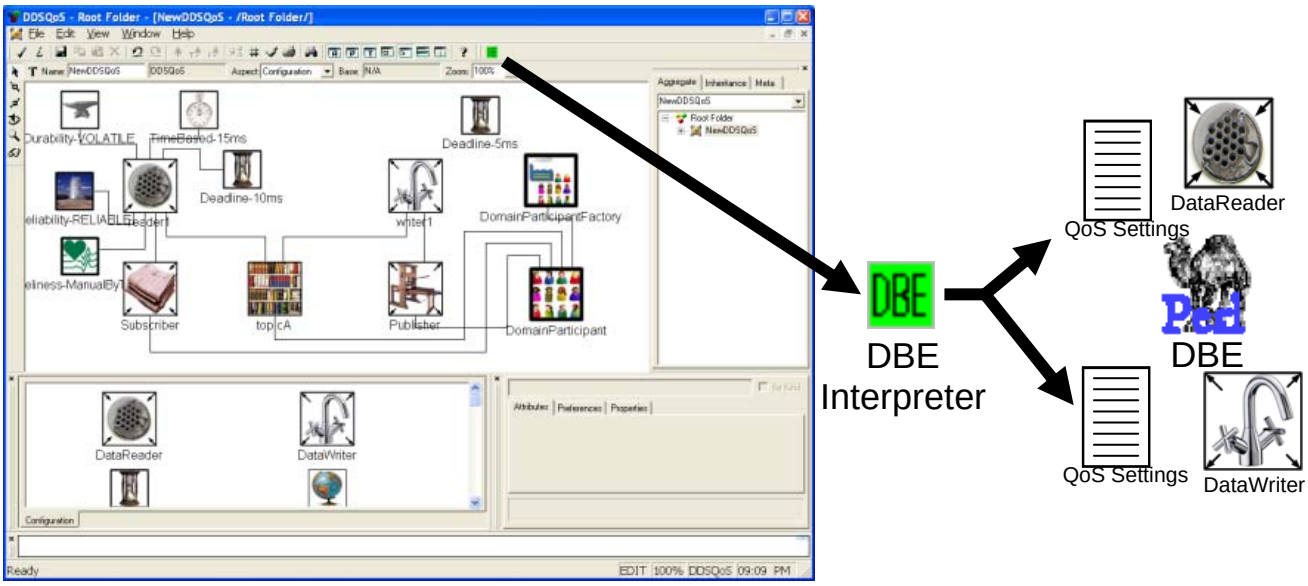


Figure 6: DQML with its DBE Interpreter

This section describes the challenges encountered when developing applications using DDS and describes how DQML addresses them.

3.1 Challenge 1: Compatibility and Consistency of QoS Settings

Problem. With the 22 QoS policies specified in DDS and the interactions between them, there is a need to ensure that QoS configurations specified for an application are compatible between different DDS entities and consistent for any one particular DDS entity. Manually checking these interactions is difficult since it is easy to miss interactions of settings that violate compatibility or consistency. Managing and changing QoS settings dynamically while the system is running adds inherent complexity and lowers the confidence level of the system. For some types of systems this lack of confidence is problematic (e.g., certain RT systems, systems with provability requirements). Iterating through the development cycle to modify source code, build, run, and test adds time and accidental complexity.

Solution. DQML has been developed to allow correct by construction configurations. At design time the developer can create a QoS configuration and check if it is compatible and consistent before the application is ever deployed.

3.2 Challenge 2: QoS Settings Generation

Problem. Generating QoS settings by hand can lead to accidental complexity. While the developer meant only to change one specific setting, other changes may inadvertently crop up (e.g., due to editing mistakes).

Solution. DQML provides a DBE interpreter that will automatically generate QoS settings files that can be used by DBE.

3.3 Challenge 3: Handling Packet Loss

Problem. Typical DDS application development includes specifying the QoS settings in the source code along with the business logic. This is a source of accidental complexity. Changes made to the source code to modify QoS settings may inadvertently modify the business logic (e.g., due to editing mistakes) since the business logic is tightly coupled with QoS configuration in the source code.

Solution. DQML decouples the business logic from QoS configuration by generating QoS settings files that can be used by the application but still remain decoupled from the source code.

Papers describing DQML and the problems it addresses appeared in the proceedings of the Distributed Objects, Middleware, and Applications (DOA'08), Monterrey, Mexico, Nov 10 - 12, 2008 and the proceedings of International Conference on Distributed Event-Based Systems (DEBS), June 20-22nd, 2007, Toronto, Canada

4.0 THE RESOURCE ALLOCATION AND CONTROL ENGINE (RACE)

We developed the *Resource Allocation and Control Engine* (RACE), which is an adaptive resource management framework built atop QoS-enabled distributed computing middleware, such as Real-time CORBA or DDS. As shown in Figure 7, RACE provides (1) *resource monitor* components that track utilization of various system resources, such as CPU, memory, and network bandwidth, (2) *QoS monitor* components that track application QoS, such as end-to-end delay, (3) *resource allocator* components that allocate resource to components based on their resource requirements and current availability of system resources, (4) *configurator* components that configure QoS parameters of application components, (5) *controller* components that compute end-to-end adaptation decisions to ensure that QoS requirements of applications are met, and (6) *effector* components that perform controller-recommended adaptations.

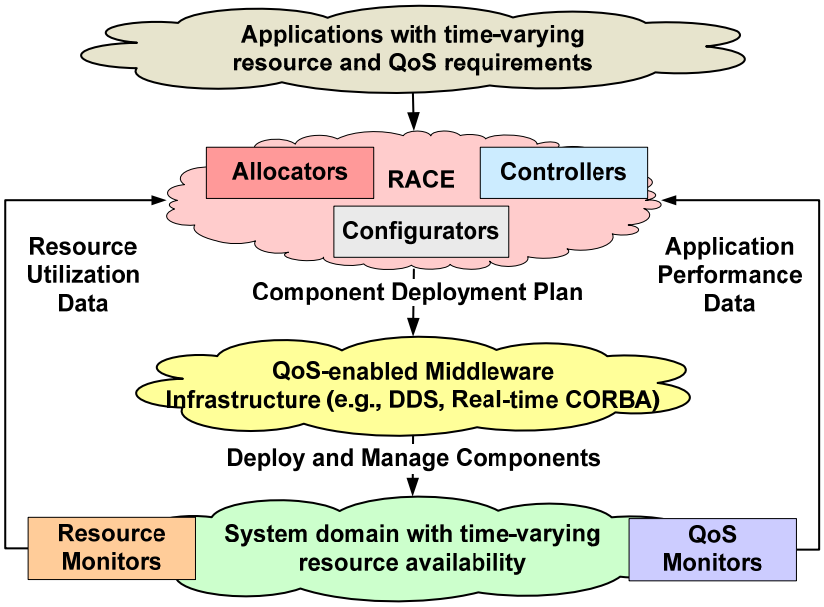


Figure 7: Resource Allocation and Control Engine (RACE) for DRE Systems

We have initially evaluated the effectiveness of RACE in the context of representative DRE systems: NASA's Magnetospheric Multi-scale Mission system (MMS) and a convey escort application. Our empirical results show that the capabilities provided by RACE yields a predictable and high performance system, even in the face of changing operational conditions, workloads, and resource availability. This section describes how RACE resolves various challenges encountered wile building a prototype implementation inspired by the MMS case study.

4.1 Challenge 1: Efficient Resource Allocation to Applications

Problem: Applications generated in the MMS system are *resource sensitive*, *i.e.*, end-to-end QoS is reduced significantly if the required type and quantity of resources are not provided to the applications at the right time. System resources should therefore be allocated in a timely fashion to components of applications such that their resource requirements are met. In open DRE systems like MMS, however, input workload affects utilization of system resources by, and QoS of, applications. These parameters of the applications may therefore vary significantly from their estimated values. Moreover, system resource availability, such as available network bandwidth, may also be time variant.

Solution. RACE monitors the current utilization of system resources and employs resource allocation algorithms, such as constraint based bin packing algorithm, to compute resource (re)allocation to applications. However, since CPU and memory utilization overhead might be associated with implementations of resource allocation algorithms themselves, RACE should, therefore, support multiple resource allocation algorithms and select the appropriate one(s) depending on properties of the application and the overheads associated with various implementations.

4.2 Challenge 2: Configuring Platform-specific QoS Parameters

Problem: QoS of applications depend on various platform-specific real-time QoS configurations including (1) QoS configuration of the QoS-enabled component middleware such as priority model, threading model, and request processing policy, (2) operating system QoS configuration such as real-time priorities of the process(es) and thread(s) that host and execute within the components respectively, and (3) networks QoS configurations, such as diffserv code-points of the component interconnections. Since these configurations are platform-specific, it is tedious and error-prone for system developers to specify them in isolation.

Solution: RACE shields application developers from low-level platform-specific details and defines a higher-level QoS specification model. Developers specify only QoS characteristics of the application, such as QoS requirements and relative importance, and RACE automatically configures platform-specific parameters accordingly.

4.3 Challenge 3: Monitoring End-to-end QoS and Ensuring QoS Requirements are Met

Problem: To meet the end-to-end QoS requirements of applications, system resource utilization and application QoS must be monitored. The system should be able to *adapt* to dynamic changes, such as variations in operational conditions, input workload, and/or resource availability, and ensure that QoS requirements of applications are not violated.

Solution: To resolve the above described challenges, RACE's control architecture employs a feedback loop to manage system resource and application QoS and ensures (1) QoS requirements of applications are met at all times and (2) system stability by maintaining utilization of system resources below their specified set-points. RACE's control architecture features a feedback loop that consists of three main components: *Monitors*, *Controllers*, and *Effectors*. Monitors tracks both system QoS and resource utilization. Controllers enable a DRE system to adapt to changing operational context and variations in resource availability and/or demand. The RACE Controllers implement various control algorithms that manage runtime system performance. Based on the control algorithm they implement, Controllers modify configurable system parameters (such as execution rates and mode of operation of the application), real-time configuration settings (such as operating system priorities of processes that host the components), and network diffserv code-points of the component interconnections.

We wrote a paper describing the structure and functionality of RACE along with our empirical evaluation that appeared in the proceedings of the 10th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing held at Santorini Island, Greece May 7-9, 2007.

5.0 MODEL DRIVEN ENGINEERING TOOLS FOR QOS CONFIGURATION

Commercial-off-the-shelf (COTS) middleware, such as application servers, QoS-enabled information grids, and object request brokers (ORBs), provide out-of-the-box support for traditional concerns affecting QoS in DRE system development, including multithreading, assigning priorities to tasks, publish/subscribe event-driven communication mechanisms, security, and multiple scheduling algorithms. This support helps decouple application logic from QoS mechanisms (such as portable priority mapping, end-to-end priority propagation, thread pools, distributable threads and schedulers, request buffering, and managing event subscriptions and event delivery necessary to support the traditional concerns listed above), shields the developers from low-level OS specific details, and promotes more effective reuse of such mechanisms.

Although component middleware has helped move the configuration complexity away from the application logic, the middleware itself has become more complex to develop and configure properly. To achieve the desired QoS characteristics for DRE systems, therefore, system developers and integrators must perform *QoS configuration* of the middleware. This process involves the binding of application level *QoS policies*—which are dictated by domain requirements—onto the solution space comprising the *QoS mechanisms* for tuning the underlying middleware. Examples of domain-level QoS policies include (1) the number of threads necessary to provide a service, (2) the priorities at which the different components should run, (3) the alternate protocols that can be used to request a service, and (4) the granularity of sharing among the application components of the underlying resources such as transport level connections.

QoS configuration bindings can be performed at several time scales, including statically, e.g., directly hard coded into the application or middleware, semi-statically, e.g., configured at deployment time using metadata descriptors, or dynamically, e.g., by modifying QoS configurations at runtime. Regardless of the binding time, however, the following challenges must be addressed:

- The need to translate the domain-specific QoS policies of the application into QoS configuration options of the underlying middleware.
- The need to choose valid values for the selected set of QoS configuration options.
- The need to understand the dependency relationships and impact between the different QoS configuration options, both at individual component level (local) as well as at aggregate intermediate levels, such as component assemblies, through the entire application (global).
- The need to validate the local and global QoS configurations, which include the values, the dependency relationships, and the semantics of QoS configuration options at all times throughout the DRE system lifecycle.

Without effective tools to address these challenges, the result will be QoS mis-configurations that are hard to analyze and debug. As a result, failures will stem from a new class of configuration errors rather than (just) traditional design/implementation errors or resource failures.

To address the QoS configuration challenges described above, we developed the Quality of service pICKER (QUICKER) model-driven engineering (MDE) toolchain shown in Figure 8. QUICKER extends the Platform-Independent Component Modeling Language (PICML), which is a domain-specific modeling language (DSML) built using the Generic Modeling Environment (GME).

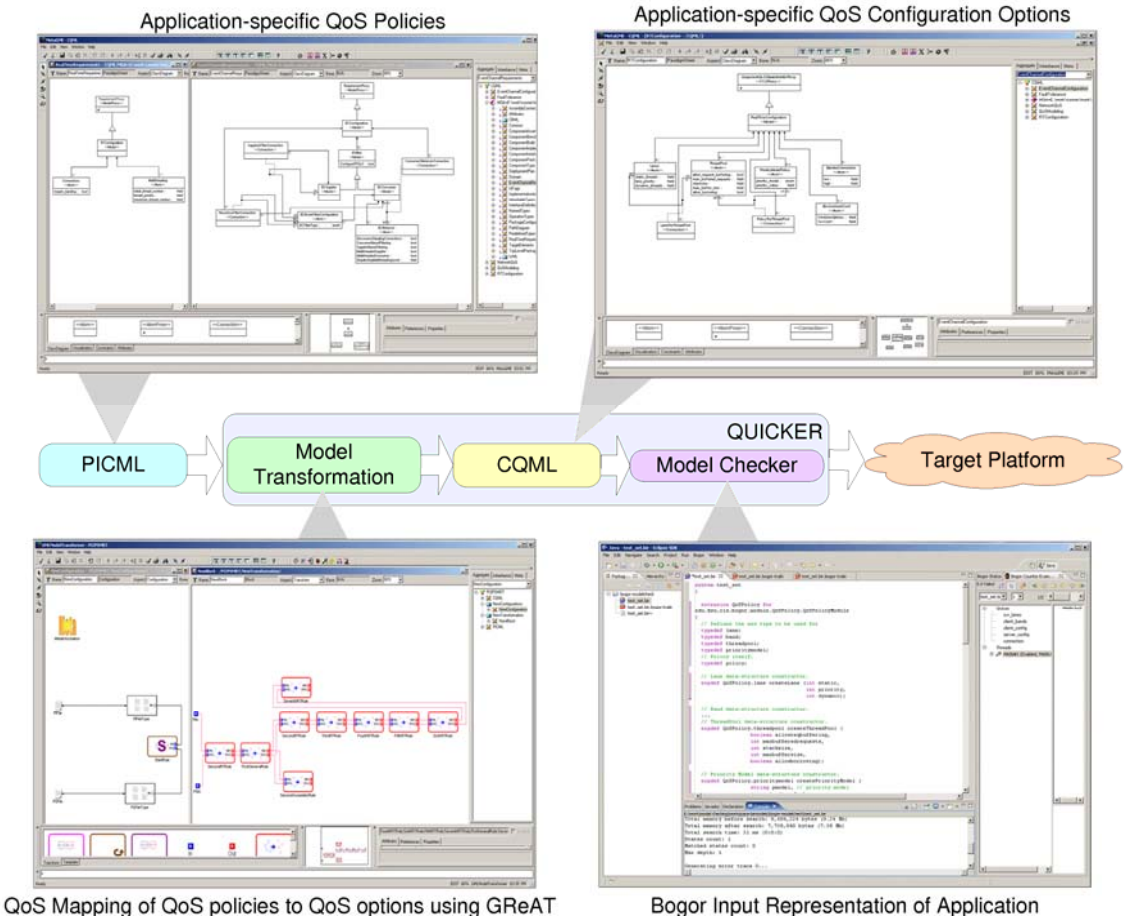


Figure 8: QUality of service pICKER (QUICKER) Toolchain

QUICKER enables developers of DRE systems to annotate applications with QoS policies. These policies are specified at a higher-level of abstraction using platform-independent models, rather than using low-level platform-specific configuration options typically found in middleware configuration files. QUICKER thus allows flexibility in binding the same QoS policy to other middleware technologies. Before the components in a DRE system can be deployed, however, their platform-independent QoS policies must be transformed into platform-specific configuration options. QUICKER therefore uses model-transformation techniques to translate the *platform-independent* specifications of QoS policies into a *platform-specific* model defined using the Component QoS Modeling Language (CQML), which models the QoS configuration options required to implement the QoS policies of the application specified in PICML. Unlike PICML (whose models are platform-independent), CQML models are specific to the underlying middleware infrastructure (which in our case is Real-time CCM).

QUICKER subsequently uses generative techniques on the CQML model to synthesize:

- The input to the Bogor model-checking framework, which validates the transformation-generated application component-specific middleware QoS configuration and identifies all permissible changes to these configuration options that can be performed at runtime, while maintaining the validity of QoS configuration across the entire application, and
- The descriptors in a middleware-specific format (such as XML) required to configure the functional and QoS properties of the application in preparation for deployment in a target environment.

This section describes the challenges in QoS configuration in middleware and how QUICKER addresses these challenges.

5.1 Challenge 1. Inherent Complexity in Translating QoS Policies to QoS Configuration Options

Problem: Translating QoS policies into QoS configuration options is hard because it must transform semantics from the application domain to the semantics of the underlying component middleware. QoS-enabled component middleware provides mechanisms to configure (1) *processor resources*, such as portable priorities, end-to-end priority propagation, thread pools, distributable threads and schedulers, (2) *communication resources*, such as protocol properties and explicit binding of connections, and (3) *memory resources*, such as buffering of requests. To translate the QoS policies into QoS mechanisms by configuring the QoS options, application developers need a thorough understanding of the underlying middleware platforms. While schedulability analysis might determine the right priority values for each component in the path of each control flow, the choice of QoS policies used to configure the middleware has a significant impact on the end result of satisfying QoS requirements. Without tool support, therefore, it is tedious and error-prone for a domain expert to translate QoS policies or analysis results to a subset of the QoS configuration options (*e.g.*, priority models, priority-bands, and thread pools) supported by the middleware that will ultimately impact the level of QoS achieved.

Solution: QUICKER gathers the application QoS policies at the domain-level abstraction and uses **model**-transformation to automate the tedious and error-prone translation of QoS policies to the appropriate subset of QoS configuration options. The *Graph Rewriting and Transformation* (GReAT) tool to transform platform-independent QoS policies captured in PICML (the input) to platform-specific QoS configuration options captured in CQML (the output).

5.2 Challenge 2. Ensuring Validity of QoS Configuration Options

Problem: Assuming that a domain expert can translate the QoS policies into a subset of QoS configuration options, it is also necessary to understand the pre-conditions, invariants, and post-conditions of the different QoS configuration options since they affect middleware behavior. This problem is exacerbated by the plethora of options and choices of valid values for each option, as well as by the fact that choosing one value for a particular option may have side effects on other options. These side effects are sometimes manifested as overt failures, such as failure to perform a mapping of CORBA priority to OS priority because of insufficient priorities in the OS to support the choice of priority mapping scheme, *e.g.*, *direct* mapping. They may also be manifested, however, as hard-to-reproduce and/or debug runtime failures that only emerge during field testing, or after deployment, which are much harder to detect and fix. In summary, validating the values of the different QoS configuration options in isolation and together with connected components is critical to the successful deployment and ultimately the operation of DRE systems. Once again, it is hard to validate these values without automated tool support.

Solution: After the model-transformation portion of the QUICKER toolchain generates a CQML model comprising the QoS configuration options, the correctness of these options must be validated before the application assembly is deployed. We validate these options using the Bogor model-checking framework, which is a customizable explicit-state model checker implemented as an Eclipse plugin.

5.3 Challenge 3. Resolving Dependencies between QoS Configuration Options

Problem: Even with a thorough understanding of middleware QoS configuration options, manual configuration of QoS policies does not scale as the number of entities to configure increases. This lack of scalability stems from dependencies between the different QoS configuration options of each component, such as the dependency between the CORBA or DDS priority of a component, the chosen priority mapping scheme (to map CORBA or DDS priority to native OS priority), and the priority-banded connections policy (which selects the appropriate connection to route requests based on the request invocation priority). As the number of components increases, the number of intra-component dependencies increases proportionally. If the components are connected, the side effect of the connection between components may also induce an inter-component option dependency. Since these dependencies can grow quadratically, it is infeasible for developers to manage these dependencies manually.

Solution: We developed Bogor Input Language (BIR) extensions to capture the interconnections between the different components in the applications. These BIR extensions were then augmented using BIR *primitives* that allowed validating the dependencies between options among connected components of an application.

5.4 Challenge 4. Ensuring Validity of QoS Configuration Options with Changes in QoS Policies

Problem: QoS configuration options effect the non-functional behavior of a system, and thus are affected by changes in the system environment. For a DRE system to operate effectively in hostile environments, such as space missions, component middleware and their associated QoS configuration options may need to adapt to their current conditions. While it is useful to change QoS configuration options at runtime to effect changes in behavior (such as re-prioritizing or increasing/decreasing resource usage), such dynamic reconfigurations may incur another set of challenges. In particular, it is non-trivial to change a running system because the system might crash during reconfiguration due to misconfiguration of QoS options. Exhaustive evaluation of possible choices of QoS configuration options and validation of the reconfigured state is too time consuming to perform at runtime and can delay the reconfiguration process itself, rendering it useless.

Solution: Our QoS extensions explore the possible states of an application and generate a set of valid application states. By exploring all the possible states of an application, QoS extensions identify both the set of valid and invalid application states. The valid states of an application can be used to select runtime QoS configurations by the RACE QoS adaptation framework at design-time. We can also construct an automaton that can guide the behavior of the RACE controller to adapt configuration options dynamically, to ensure that reconfiguration will not yield an invalid application state.

We published a paper describing the structure and functionality of QUICKER along with our empirical evaluation called that appeared at the 10th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing held at Santorini Island, Greece May 7-9, 2007.

6.0 RICOCHET++ ADAPTIVE MIDDLEWARE/TRANSPORT FRAMEWORK

We collaborated with Cornell University on the Ricochet++ project, which integrates the OpenDDS QoS-enabled middleware with the Ricochet transport protocol developed by Ken Birman's group as part of the Castor project. OpenDDS is an open-source implementation of the Data Distribution Service (DDS) that supports a pluggable transport framework. This framework allows transport protocols to be used by OpenDDS for data transport. OpenDDS is open-source software supported by Object Computing, Inc. (www.ociweb.com). Ricochet provides low latency loss detection, low latency loss correction for single missed packets, and probabilistic reliability. It is built on top of IP multicast and requires no modifications to routers or gateways. All the functionality provided by Ricochet is managed in the end hosts. Figure 9 below illustrates the behavior of Ricochet when there are no dropped packets. Error correction information is passed between the receivers of the multicast data similar in concept to gossip-based protocols.

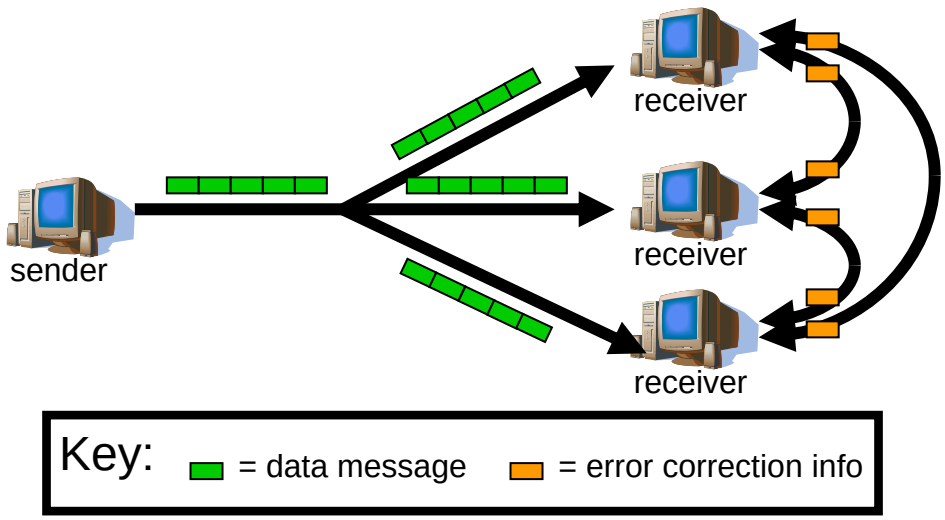


Figure 9: Ricochet Running Forward Error Correction (FEC) Algorithm

Figure 10 below shows the behavior of Ricochet when a data packet is dropped. This condition is detected quickly between receivers since they exchange information as to what packets they have already received. Additionally, this information also allows receivers to reconstruct a lost packet. We initially downloaded the Ricochet source code and started running the example application supplied to gain familiarity with Ricochet. We then investigated OpenDDS's pluggable transport framework to understand how to implement the integration of OpenDDS and Ricochet. In addition, we determined how to interface Java and C++ since OpenDDS is written in C++ and the Ricochet transport protocol is written in Java.

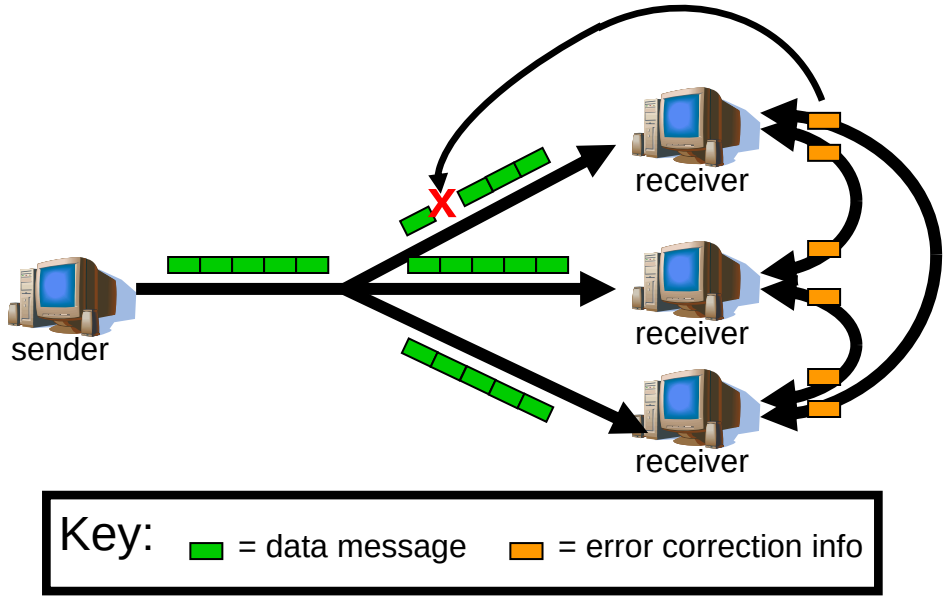


Figure 10: Ricochet Using Forward Error Correction To Correct Packet Loss

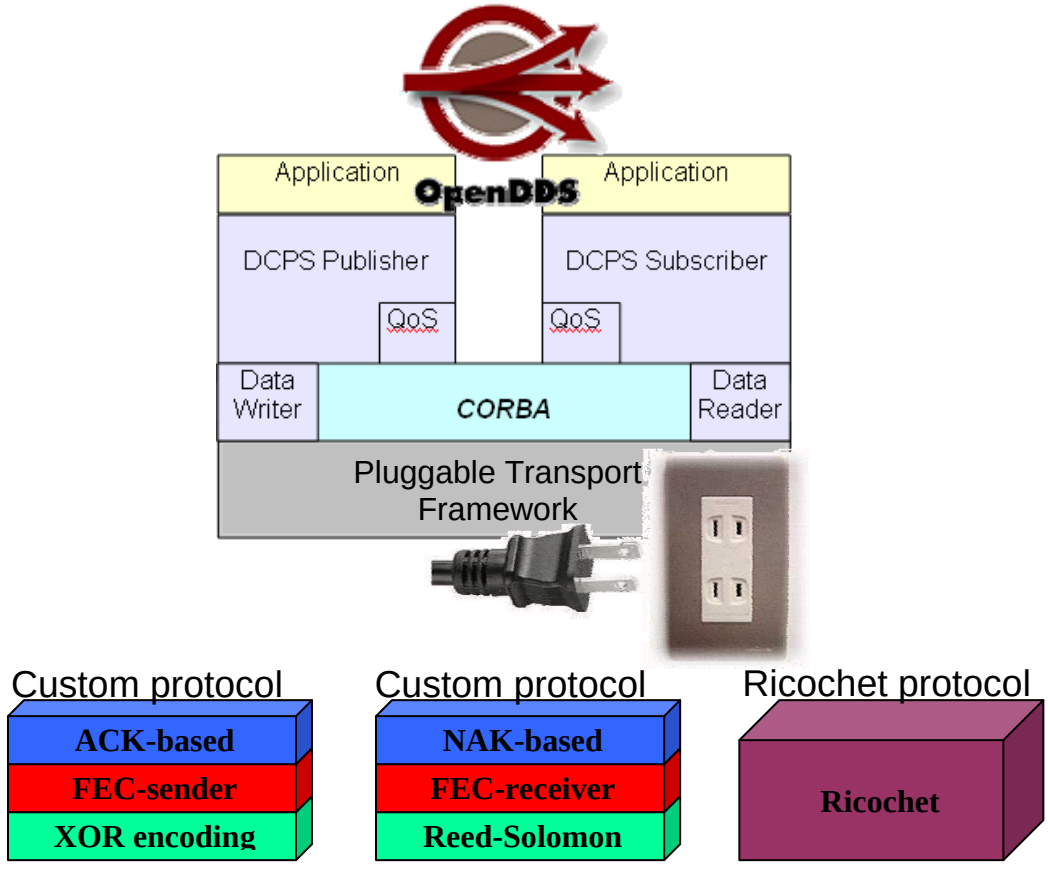


Figure 11: OpenDDS and Ricochet Integration

We enhanced the OpenDDS and Ricochet++ integration shown in Figure 11 to support multiple topics where each topic will be mapped to a multicast group in Ricochet++. Previously only a single Ricochet++ multicast group was supported for any one OpenDDS executable within the distributed metrics application.Additionally, we developed software to collect latency metrics for an OpenDDS application with one sender and multiple receivers. The publisher sends out data on multiple topics which correlate to multiple multicast groups within Ricochet++. A subscriber then receives the data for the topic to which it subscribed and calculates latency metrics for the data.

We presented a poster entitled “Trustworthy Conferencing via Domain-specific Modeling and Low Latency Reliable Protocols” at the NSF TRUST Spring 2008 Conference in April. We also published papers on our Ricochet++ work at the 2nd workshop on Large-Scale Distributed Systems and Middleware (LADIS 2008), IBM TJ Watson Research Center, Yorktown, New York, September 2008.

7.0 CUTS SYSTEM EXECUTION MODELING TOOL ENHANCEMENTS

The *Component Workload Emulator (Co-WorkEr) Utilization Test Suite (CUTS)* is a system execution modeling (SEM) tool that helps developers conduct "what if" experiments to discover, measure, and rectify performance problems *early* in the lifecycle (e.g., in the architecture and design phases), as opposed to the integration phase, when mistakes are much harder and more costly to fix. In particular, CUTS provides the following capabilities:

1. It allows users (e.g., software architects, developers, and systems engineers) specify the structure of an enterprise DRE system (e.g., the component and their interconnections us-ing CUTS DSMLs).
2. It allows users to associate the necessary QoS characteristics with individual compo-nents (e.g., CPU utilization) or the system as a whole (e.g., deadline of a critical path through the system).
3. It allows the information captured by the tools can be synthesized into executable code and configuration metadata, which the middleware then uses to deploy the emu-lated/actual application/system components onto the target platform.
4. It allows system developers and engineers to analyze the collected metrics and explore design alternatives from multiple computational and valuation perspectives to quantify the costs of certain design choiceson end-to-end system performance.

This process can be applied iteratively throughout the phases of development process.

In the Pollux project we developed a data collection and analysis framework used when integrating the CUTS system execution modeling tool with continuous integration environments, such as CruiseControl (ccnet.throughworks.com), to create CiCUTS, and began the first phase of identifying search algorithms for locating deployments and configuration of component-based system that meet desired performance requirements.

The motivation for generalizing the data collection and analysis framework for CiCUTS is to improve its applicability across multiple project domains. Moreover, provide capabilities to large-scale distributed system developers who collect large amounts of varying data for understanding system performance at multiple levels without having to deal with the complexity to implementing an analysis and reporting framework. By using CiCUTS's generic reporting and analysis framework, developers are be able to identify metrics of interest at a high-level, such as throughput of an event, or test and test configurations, and let CiCUTS monitor and analyze such metrics continuously throughout the system's development lifecycle.

```
…
    CiCUTS.instance ().log (LM_INFO, “received ” + events + “ events”);
…
```

Listing 1. Example Log Message for Capturing Performance

As illustrated in Listing 1, developers use simple log messages to identify their metrics of interest and embed the messages in their source code. We use log messages because they (1) are flexible enough to capture any arbitrary performance metric, (2) can be inserted/removed quickly to modify collected performance metrics, (3) serve as good indicators for understanding behavior and performance.

```
``received {INT x} events’’
```

Listing 2. Example High-level Regular Expression for Analyzing Performance

From a testing standpoint, capturing the performance metrics will be handled seamlessly by the CiCUTS’s testing environment. Developers, therefore, only have to identify what metrics need to be analyzed by the analysis and reporting framework. Listing 2 shows an example regular expression that corresponds to the performance metric presented in Listing 1. For each test identified in the database, the analysis framework will convert the high-level regular expressions into usable regular expressions for data mining against collected log messages. From the extracted log messages, it will present a graph of the identified performance metric.

```
…
CiCUTS.instance ().log (LM_INFO, “received ” + events + “ events over a ” + duration + “
                                    minute test”);
…

``received {INT x} events over a {INT duration} minute test’’
                    analysis: x / duration
```

Listing 3. High-level Regular Expression with On-The-Fly Evaluation

For cases where a single identifier is insufficient to capture performance, we provide on-the-fly evaluation. As highlighted in Listing 3, the log message captures the number of events published and the duration of the test. If we want to capture how many events were published per second, but failed to do the calculation when creating the log message, we can create an equation that evaluates the identifiers in the log message.

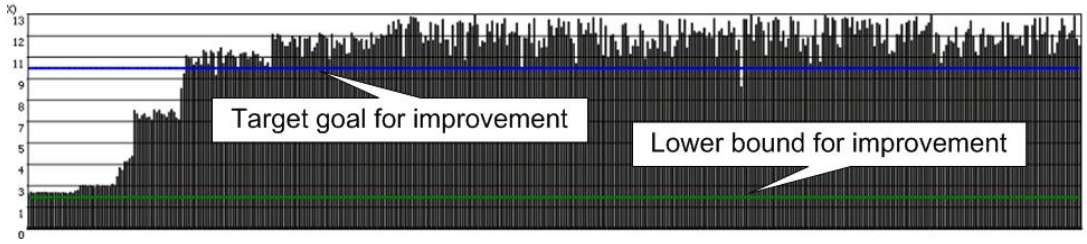


Figure 12: Analysis of a Single Unit Test for Multiple Test Runs

Once the high-level metrics have been data mined and analyzed, results for the test run are displayed to the tester. In addition to showing metrics for single test run, the analysis framework is able to show metrics for multiple tests runs, such as a night build of the system. As illustrated in Figure 12, the analysis framework presents the results of a single unit test for multiple test runs. This is similar to the initial effort; however, it extends that effort because all results are generated from metrics that have been identified at a high-level.